**Experiment : 1**

**Title: To Study various Software Testing and Quality Assurance Tools**

**Theory:**

**Software Testing:**

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include, but are not limited to the process of executing a program or application with the intent of finding software bugs (errors or other defects).

Software testing can be stated as the process of validating and verifying that a computer program/application/product:

* meets the requirements that guided its design and development,
* works as expected,
* can be implemented with the same characteristics,
* and satisfies the needs of stakeholders.

Software testing, depending on the testing method employed, can be implemented at any time in the software development process. Traditionally most of the test effort occurs after the requirements have been defined and the coding process has been completed, but in the Agile approaches most of the test effort is on-going. As such, the methodology of the test is governed by the chosen software development methodology.

**Objectives of testing:**

* To discuss the distinctions between validation testing and defect testing
* To describe the principles of system and component testing
* To describe strategies for generating system test cases
* To understand the essential characteristics of tool used for test automation
* Executing a program with the intent of finding an *error*.
* To check if the system meets the requirements and be executed successfully in the Intended environment.
* To check if the system is “Fit for purpose”.

**Software Quality assurance Tools**

**Theory:**

Quality - the most important factor affecting an organization’s long-term performance.

Software quality assurance (SQA) consists of a means of monitoring the software engineering processes and methods used to ensure quality. The methods by which this is accomplished are many and varied, and may include ensuring conformance to one or more standards, such as ISO 9000 or a model such as CMMI.

SQA encompasses the entire software development process, which includes processes such as requirements definition, software design, coding, source code control, code reviews, change management, configuration management, testing, release management, and product integration. SQA is organized into goals, commitments, abilities, activities, measurements, and verifications.

**What is Software Quality Assurance?**

* Used to Monitor and Improve the Software Development Process
* Making Sure That Standards and Procedures are Followed
* Ensures that Problems are Found and Dealt with
* Orientated to ‘Prevention

**Techniques:**

Audit:

* The Major Technique used in SQA
* Perform Product Evaluation and Process Monitoring
* Performed Routinely throughout the Software Development Process
* Look at a Process and/or Product in depth and compare to Established Standards and Procedures
* Purpose is to assure that:
  + Proper Control Procedures are being followed
  + Required Documentation is maintained
  + Developer’s Status Reports accurately reflect the status of the activity
* Used to:

**Experiment : 8**

**Title: To Test a Web Based System**

**Theory:**

Technically, the term Web-Based system refers to those applications or services that are resident on a server that is accessible using a Web browser and is therefore accessible from anywhere in the world via the Web. Web based applications are the ultimate way to take advantage of today's technology to enhance your organizations productivity & efficiency. Web based application gives you an opportunity to access your business information from anywhere in the world at any time. Web based applications have come a long way and now offer competitive advantages to traditional software based systems allowing businesses to consolidate and streamline their systems and processes

They offer Cross platform compatibility are more manageable, highly deployable, give secure live data. It also facilitates you to save time and money and improve the interactivity with your customers and partners.

Web based applications have evolved significantly over recent years and with improvements in security and technology there are plenty of scenarios where traditional software based applications and systems could be improved by migrating them to a web based application. We at IRES have the expertise to implement all these scenarios for your business.

Web testing is the name given to [software testing](http://en.wikipedia.org/wiki/Software_testing) that focuses on [web applications](http://en.wikipedia.org/wiki/Web_application). Complete testing of a web-based system before going live can help address issues before the system is revealed to the public. Issues such as the security of the web application, the basic functionality of the site, its accessibility to handicapped users and fully able users, as well as readiness for expected traffic and number of users and the ability to survive a massive spike in user traffic, both of which are related to [load testing](http://en.wikipedia.org/wiki/Load_testing).

**Few Steps that can be followed while testing a Website:**

**• Functionality Testing  
• Performance Testing  
• Usability Testing  
• Server Side Interface  
• Client Side Compatibility  
• Security**

**Functionality:**  
 In testing the functionality of the web sites the following should be tested:  
**• Links**

i. Internal Links

**Experiment : 5**

**Title : To find Cyclomatic Complexity of a program using different methods.**

**Theory :**

Cyclomatic complexity measures the amount of decision logic in a single software module. It is used for two related purposes in the structured testing methodology.

First, it gives the number of recommended number of independent paths from entry to exit in programming element or its representation. . Second, it is used during all phases of the software lifecycle, beginning with design, to keep software reliable, testable, and manageable

Cyclomatic complexity is based entirely on the structure of software's control flow graph. For each module, Cyclomatic complexity is defined to be

Cyclomatic complexity, v(G) = e - n + 2

where

v refers to the cyclomatic number in graph theory

G indicates that the complexity is a function of the graph

e represents number of edges

n represents nodes in the control flow graph

The cyclomatic number in graph theory is defined as e - n + 1. Program control flow graphs are not strongly connected, but they become strongly connected when a "virtual edge" is added connecting the exit node to the entry node. The cyclomatic complexity definition for program control flow graphs is derived from the cyclomatic number formula by simply adding one to represent the contribution of the virtual edge. This definition makes the cyclomatic complexity equal the number of independent paths through the standard control flow graph model, and avoids explicit mention of the virtual edge.

**Experiment : 2**

**Title : Debugging capabilities of C/C++ IDEs.**

**Theory :**

C and C++ programs compiled with the GNU compiler and the -g option can be debugged using GNU's debugger gdb (actually, you can use gdb on code that is not compiled with -g, but unless you like trying to figure out how assembly code sequences map to your source code I wouldn't recommend doing so). Also, do not compile with an optimization flag (i.e. don't use -O2), or gdb will have a hard time mapping optimized machine code to your source code.

One good way to get started when you are trying to track down a bug, is to set breakpoints at the start of every function. In this way, you will quickly be able to determine which function has the problem. Then you can restart the program and step through the offending function line-by-line until you locate the problem exactly.

When debugging lines of code, here are the usual scenarios:

* (Step Into) A method is about to be invoked, and you want to debug into the code of that method, so the next step is to go into that method and continue debugging step-by-step.
* (Step Over) A method is about to be invoked, but you're not interested in debugging this particular invocation, so you want the debugger to execute that method completely as one entire step.
* (Step Return) You're done debugging this method step-by-step, and you just want the debugger to run the entire method until it returns as one entire step.
* (Resume) You want the debugger to resume "normal" execution instead of step-by-step
* (Line Breakpoint) You don't care how it got there, but if execution reaches a particular line of code, you want the debugger to temporarily pause execution there so you can decide what to do.

**Breakpoint**

In software development, a breakpoint is an intentional stopping or pausing place in a program, put in place for debugging purposes. It is also sometimes simply referred to as a pause.

More generally, a breakpoint is a means of acquiring knowledge about a program during its execution. During the interruption, the programmer inspects the test environment (general purpose registers, memory, logs, files, etc.) to find out whether the program is functioning as expected. In practice, a breakpoint consists of one or more conditions that determine when a program's execution should be interrupted.

**Breakpoint conditions**

The most common form of a breakpoint is the one where the program's execution is interrupted right before a programmer-specified [instruction](http://en.wikipedia.org/wiki/Instruction_(computer_science)) is executed. This is often referred to as an instruction breakpoint.

Other kinds of conditions can also be used, such as the reading, writing, or modification of a specific location in an area of memory. This is often referred to as a conditional breakpoint, a data breakpoint, or a watchpoint.

Breakpoints can also be used to interrupt execution at a particular time, upon a keystroke etc.

**Inspection tools**

When a breakpoint is hit, various tools are used to inspect the state of the program or alter it. [Stack trace](http://en.wikipedia.org/wiki/Stack_trace) of each [thread](http://en.wikipedia.org/wiki/Thread_(computer_science)) may be used to see the chain of [function](http://en.wikipedia.org/wiki/Subroutine) calls that led to the paused instruction. A list of watches allows to view the values of selected [variables](http://en.wikipedia.org/wiki/Variable_(programming)) and [expressions](http://en.wikipedia.org/wiki/Expression_(programming)). There may also be tools to show the contents of [registers](http://en.wikipedia.org/wiki/Processor_register), loaded program [modules](http://en.wikipedia.org/wiki/Module_(programming)) and other information.

**Implementations**

**Hardware**

Many [processors](http://en.wikipedia.org/wiki/Central_processing_unit) include [hardware](http://en.wikipedia.org/wiki/Computer_hardware) support for breakpoints (typically instruction and data breakpoints). As an example, the x86 instruction set architecture provides hardware support for breakpoints with its [x86 debug registers](http://en.wikipedia.org/wiki/X86_debug_register). Such hardware may include limitations, for example not allowing breakpoints on instructions located in [branch delay slots](http://en.wikipedia.org/wiki/Branch_delay_slot). This kind of limitation is imposed by the [microarchitecture](http://en.wikipedia.org/wiki/Microarchitecture) of the processor and varies from processor to processor.

**Software**

Without hardware support, [debuggers](http://en.wikipedia.org/wiki/Debugger) have to implement breakpoints in software. For instruction breakpoints, this is a comparatively simple task of replacing the instruction at the location of the breakpoint by either:

* an instruction that calls the debugger directly (e.g. a [system call](http://en.wikipedia.org/wiki/System_call)) or
* an invalid instruction that causes a deliberate program interrupt (that is then intercepted/handled by the debugger)

Alternatively,

* an [instruction set simulator](http://en.wikipedia.org/wiki/Instruction_set_simulator) can implement unconditional or conditional breakpoints, by simply embedding the appropriate condition tests within its own normal [program cycle](http://en.wikipedia.org/wiki/Main_loop) - that also naturally allows non-invasive breakpoints (on [read-only](http://en.wikipedia.org/wiki/Read-only_memory) programs for instance).
* [Interpreted languages](http://en.wikipedia.org/wiki/Interpreted_language) can effectively use the same concept as above in their program cycle.
* ["Instrumenting"](http://en.wikipedia.org/wiki/Instrumentation_(computer_programming)) all the source code with additional source statements that issue a [function](http://en.wikipedia.org/wiki/Function_(computer_science)) that invoke an internal or external debug subroutine, is yet another common approach. This method increases the [binary](http://en.wikipedia.org/wiki/Binary_file) size and might adversely affect normal memory allocation and [exception handlers](http://en.wikipedia.org/wiki/Exception_handler). "Debug" options exist on some compilers to implement this technique semi-transparently.

Some debuggers allow program variables in memory to be modified before resuming, effectively allowing the introduction of "hand-coded" temporary assignments for test purposes. Similarly, program instructions can often be skipped to determine the effect of changes to the program logic - enabling questions about program execution to be answered in a direct way (i.e. without assumptions or guesswork). In many cases it may be the only practical method of testing obscure "event-driven" error subroutines that rarely, if ever, get executed - without the added risk of leaving temporary source changes.

Implementing data breakpoints in software however, can greatly reduce the performance of the application being debugged - since it is using additional resources on the same processor. However, this is normally acceptable during testing and the amount of information available from the debugger is not restricted by limitations of debug data known to the hardware. For instance, a software implementation can collect logical path data at program/subroutine/instruction level to considerably augment what might be stored by the particular hardware platform for inspection. The instruction set simulation method considerably reduces the compared to the (repeated) instruction replacement method, also reducing [cache misses](http://en.wikipedia.org/wiki/Cache_miss).

Some programming language implementations [expose](http://en.wikipedia.org/wiki/Reflection_(computer_science)) their debugging functions for use by other programs. For example, some [FORTRAN](http://en.wikipedia.org/wiki/FORTRAN) dialects have an AT statement, which was originally intended to act as an instruction breakpoint. [Python](http://en.wikipedia.org/wiki/Python_(programming_language)) implements a debugger accessible from a Python program. These facilities can be and are abused to act like the [COMEFROM](http://en.wikipedia.org/wiki/COMEFROM) statement.

**Step Over**

Select the **Step Over** command to step over the next method call (without entering it) at the currently executing line of code. Even though the method is never stepped into, the method will be executed normally.

To step over a method you must have execution suspended and be stepping through code.

**Step Into**

Select the **Step Into** command to step into the next method call at the currently executing line of code.

To step into a method you must have execution suspended and be stepping through code.

**Watch Window**

The Watch window and QuickWatch dialog box are places where you can enter variable names and expressions that you want to watch during a debugging session.

The QuickWatch dialog box enables you to examine a single variable or expression at a time. It is useful for taking a quick look at one value or a larger data structure. The Watch window can store several variables and expressions that you want to view over the course of the debugging session. Some editions of Visual Studio have multiple Watch windows, which are numbered Watch1 through Watch4.

A variable name is the simplest expression you can enter. If you are debugging native code, you can use register names as well as variable names. The debugger can accept much more complex expressions than that, however. For example, you could enter the following expression to find the average value of three variables:

(var1 + var2 + var3) / 3

**Experiment : 3**

**Title : Use of a Software Testing tool: CPPUnit**

**Theory :**

The Cpp Unit test framework is for unit test of C++ class functions. It relies on the hierarchy of a test suite comprising of unit test cases which test class functions. The test begins with setUp() followed by the test and ending with tearDown().

Each unit test employs the use of C++ assert() to test the function results.

C++ assert prototype: void assert (int expression);

If this expression evaluates to 0, this causes an assertion failure that terminates the program.

The assert function will abort the application if false.

Tests in CppUnit can be run automatically. They are easy to set up and once you have written them, they are always there to help you keep confidence in the quality of your code.

**To make a simple test, here is what you do:**

Subclass the [TestCase](http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html)class. Override the method [runTest()](http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html). When you want to check a value, call [CPPUNIT\_ASSERT(bool)](http://cppunit.sourceforge.net/doc/lastest/group___assertions.html#ga0)and pass in an expression that is true if the test succeeds.

For example, to test the equality comparison for a Complex number class, write:

class ComplexNumberTest : public CppUnit::TestCase

{

public:

ComplexNumberTest( std::string name ) : CppUnit::TestCase( name )

{}

void runTest()

{

CPPUNIT\_ASSERT( Complex (10, 1) == Complex (10, 1) );

CPPUNIT\_ASSERT( !(Complex (1, 1) == Complex (2, 2)) );

}

};

That was a very simple test. Ordinarily, you'll have many little test cases that you'll want to run on the same set of objects. To do this, use a fixture.

**Fixture**

A fixture is a known set of objects that serves as a base for a set of test cases. Fixtures come in very handy when you are testing as you develop.

Let's try out this style of development and learn about fixtures along the away. Suppose that we are really developing a complex number class. Let's start by defining a empty class named Complex.

class Complex {};

Now create an instance of ComplexNumberTest above, compile the code and see what happens. The first thing we notice is a few compiler errors. The test uses operator ==, but it is not defined. Let's fix that.

bool operator = = ( const Complex &a, const Complex &b)

{

return true;

}

Now compile the test, and run it. This time it compiles but the test fails. We need a bit more to get an operator == working correctly, so we revisit the code.

class Complex

{

friend bool operator ==(const Complex& a, const Complex& b);

double real, imaginary;

public:

Complex( double r, double i = 0 ) : real(r), imaginary(i)

{ }

};

bool operator ==( const Complex &a, const Complex &b )

{

return a.real == b.real && a.imaginary == b.imaginary;

}

If we compile now and run our test it will pass.

Now we are ready to add new operations and new tests. At this point a fixture would be handy. We would probably be better off when doing our tests if we decided to instantiate three or four complex numbers and reuse them across our tests.

Here is how we do it:

* Add member variables for each part of the [fixture](http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html)
* Override [setUp()](http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html)to initialize the variables
* Override [tearDown()](http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html)to release any permanent resources you allocated in [setUp()](http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html)

class ComplexNumberTest : public CppUnit::TestFixture

{

private:

Complex \*m\_10\_1, \*m\_1\_1, \*m\_11\_2;

public:

void setUp()

{

m\_10\_1 = new Complex( 10, 1 );

m\_1\_1 = new Complex( 1, 1 );

m\_11\_2 = new Complex( 11, 2 );

}

void tearDown()

{

delete m\_10\_1;

delete m\_1\_1;

delete m\_11\_2;

}

};

Once we have this fixture, we can add the complex addition test case any any others that we need over the course of our development.

**Test Case**

How do you write and invoke individual tests using a fixture?

There are two steps to this process:

* Write the test case as a method in the fixture class
* Create a [TestCaller](http://cppunit.sourceforge.net/doc/lastest/class_test_caller.html) which runs that particular method

Here is our test case class with a few extra case methods:

class ComplexNumberTest : public CppUnit::TestFixture

{

private:

Complex \*m\_10\_1, \*m\_1\_1, \*m\_11\_2;

public:

void setUp()

{

m\_10\_1 = new Complex( 10, 1 );

m\_1\_1 = new Complex( 1, 1 );

m\_11\_2 = new Complex( 11, 2 );

}

void tearDown()

{

delete m\_10\_1;

delete m\_1\_1;

delete m\_11\_2;

}

void testEquality()

{

CPPUNIT\_ASSERT( \*m\_10\_1 == \*m\_10\_1 );

CPPUNIT\_ASSERT( !(\*m\_10\_1 == \*m\_11\_2) );

}

void testAddition()

{

CPPUNIT\_ASSERT( \*m\_10\_1 + \*m\_1\_1 == \*m\_11\_2 );

}

};

One may create and run instances for each test case like this:

CppUnit::TestCaller<ComplexNumberTest> test( "testEquality", &ComplexNumberTest::testEquality );

CppUnit::TestResult result;

test.run( &result );

The second argument to the test caller constructor is the address of a method on ComplexNumberTest. When the test caller is run, that specific method will be run. This is not a useful thing to do, however, as no diagnostics will be displayed. One will normally use a [TestRunner](http://cppunit.sourceforge.net/doc/lastest/group___executing_test.html)(see below) to display the results.

Once you have several tests, organize them into a suite.

**Suite**

How do you set up your tests so that you can run them all at once?

CppUnit provides a [TestSuite](http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html)class that runs any number of TestCases together.

We saw, above, how to run a single test case.

To create a suite of two or more tests, you do the following:

CppUnit::TestSuite suite;

CppUnit::TestResult result;

suite.addTest( new CppUnit::TestCaller<ComplexNumberTest>("testEquality", &ComplexNumberTest::testEquality ) );

suite.addTest( new CppUnit::TestCaller<ComplexNumberTest>("testAddition", &ComplexNumberTest::testAddition ));

suite.run( &result );

[TestSuites](http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html)don't only have to contain callers for TestCases. They can contain any object that implements the [Test](http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html)interface. For example, you can create a [TestSuite](http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html)in your code and I can create one in mine, and we can run them together by creating a [TestSuite](http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html)that contains both:

CppUnit::TestSuite suite;

CppUnit::TestResult result;

suite.addTest( ComplexNumberTest::suite() );

suite.addTest( SurrealNumberTest::suite() );

suite.run( &result );TestRunner

**How do you run your tests and collect their results?**

Once you have a test suite, you'll want to run it. CppUnit provides tools to define the suite to be run and to display its results. You make your suite accessible to a [TestRunner](http://cppunit.sourceforge.net/doc/lastest/group___executing_test.html)program with a static method *suite* that returns a test suite.

For example, to make a ComplexNumberTest suite available to a [TestRunner](http://cppunit.sourceforge.net/doc/lastest/group___executing_test.html), add the following code to ComplexNumberTest:

public:

static CppUnit::Test \*suite()

{

CppUnit::TestSuite \*suiteOfTests = new CppUnit::TestSuite( "ComplexNumberTest" );

suiteOfTests->addTest( new CppUnit::TestCaller<ComplexNumberTest>("testEquality", &ComplexNumberTest::testEquality ) );

suiteOfTests>addTest(newCppUnit::TestCaller<ComplexNumberTest>("testAddition", &ComplexNumberTest::testAddition ) );

return suiteOfTests;

}

To use the text version, include the header files for the tests in Main.cpp:

#include <cppunit/ui/text/TestRunner.h>

#include "ExampleTestCase.h"

#include "ComplexNumberTest.h"

And add a call to addTest(CppUnit::Test \*) in the main() function:

int main( int argc, char \*\*argv)

{

CppUnit::TextUi::TestRunner runner;

runner.addTest( ExampleTestCase::suite() );

runner.addTest( ComplexNumberTest::suite() );

runner.run();

return 0;

}

The [TestRunner](http://cppunit.sourceforge.net/doc/lastest/group___executing_test.html)will run the tests. If all the tests pass, you'll get an informative message. If any fail, you'll get the following information:

* The name of the test case that failed
* The name of the source file that contains the test
* The line number where the failure occurred
* All of the text inside the call to [CPPUNIT\_ASSERT()](http://cppunit.sourceforge.net/doc/lastest/group___assertions.html#ga0) which detected the failure

CppUnit distinguishes between *failures* and *errors*. A failure is anticipated and checked for with assertions. Errors are unanticipated problems like division by zero and other exceptions thrown by the C++ runtime or your code.

**Helper Macros**

As you might have noticed, implementing the fixture static suite() method is a repetitive and error prone task. A [Writing test fixture](http://cppunit.sourceforge.net/doc/lastest/group___writing_test_fixture.html) set of macros have been created to automatically implements the static suite() method.

The following code is a rewrite of ComplexNumberTest using those macros:

#include <cppunit/extensions/HelperMacros.h>

class ComplexNumberTest : public CppUnit::TestFixture

{

First, we declare the suite, passing the class name to the macro:

CPPUNIT\_TEST\_SUITE( ComplexNumberTest );

The suite created by the static suite() method is named after the class name. Then, we declare each test case of the fixture:

CPPUNIT\_TEST( testEquality );

CPPUNIT\_TEST( testAddition );

Finally, we end the suite declaration:

CPPUNIT\_TEST\_SUITE\_END();

At this point, a method with the following signature has been implemented:

static CppUnit::TestSuite \*suite();

The rest of the fixture is left unchanged:

private:

Complex \*m\_10\_1, \*m\_1\_1, \*m\_11\_2;

public:

void setUp()

{

m\_10\_1 = new Complex( 10, 1 );

m\_1\_1 = new Complex( 1, 1 );

m\_11\_2 = new Complex( 11, 2 );

}

void tearDown()

{

delete m\_10\_1;

delete m\_1\_1;

delete m\_11\_2;

}

void testEquality()

{

CPPUNIT\_ASSERT( \*m\_10\_1 == \*m\_10\_1 );

CPPUNIT\_ASSERT( !(\*m\_10\_1 == \*m\_11\_2) );

}

void testAddition()

{

CPPUNIT\_ASSERT( \*m\_10\_1 + \*m\_1\_1 == \*m\_11\_2 );

}

};

The name of the [TestCaller](http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html)added to the suite are a composition of the fixture name and the method name.

In the present case, the names would be: "ComplexNumberTest.testEquality" and "ComplexNumberTest.testAddition".

The [helper macros](http://cppunit.sourceforge.net/doc/lastest/group___writing_test_fixture.html)help you write comon assertion. For example, to check that ComplexNumber throws a MathException when dividing a number by 0:

* add the test to the suite using CPPUNIT\_TEST\_EXCEPTION, specifying the expected exception type.
* write the test case method

CPPUNIT\_TEST\_SUITE( ComplexNumberTest );

// [...]

CPPUNIT\_TEST\_EXCEPTION( testDivideByZeroThrows, MathException );

CPPUNIT\_TEST\_SUITE\_END();

void testDivideByZeroThrows()

{

// The following line should throw a MathException.

\*m\_10\_1 / ComplexNumber(0);

}

If the expected exception is not thrown, then a assertion failure is reported.

**Post-build check**

Well, now that we have our unit tests running, how about integrating unit testing to our build process ?

To do that, the application must returns a value different than 0 to indicate that there was an error.

TestRunner::run() returns a boolean indicating if the run was successful.

Updating our main programm, we obtains:

#include <cppunit/extensions/TestFactoryRegistry.h>

#include <cppunit/ui/text/TestRunner.h>

int main( int argc, char \*\*argv)

{

CppUnit::TextUi::TestRunner runner;

CppUnit::TestFactoryRegistry &registry = CppUnit::TestFactoryRegistry::getRegistry();

runner.addTest( registry.makeTest() );

bool wasSuccessful = runner.run( "", false );

return wasSuccessful;

}

**Experiment : 4**

**Title : Write a program to find number of loops and decision making statements in a C/C++ program.**

**Theory :**

**Loops:**

Loops are used to repeat a block of code. Being able to have your program repeatedly execute a block of code is one of the most basic but useful tasks in programming - many programs or websites that produce extremely complex output (such as a message board) are really only executing a single task many times. (They may be executing a small number of tasks, but in principle, to produce a list of messages only requires repeating the operation of reading in some data and displaying it.) Now, think about what this means: a loop lets you write a very simple statement to produce a significantly greater result simply by repetition.

There are three types of loops: for, while, and do..while. Each of them has their specific uses. They are all outlined below.

**FOR** - for loops are the most useful type. The syntax for a for loop is

for ( variable initialization; condition; variable update )

{

Code to execute while the condition is true

}

**WHILE** - WHILE loops are very simple. The basic structure is

while ( condition )

{

Code to execute while the condition is true

}

**DO..WHILE** - DO..WHILE loops are useful for things that want to loop at least once. The structure is

do

{

} while ( condition );

**Decision making Statements:**

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages:



|  |  |
| --- | --- |
| **Statement** | **Description** |
| [**if statement**](http://www.tutorialspoint.com/cprogramming/if_statement_in_c.htm) | **An if statement consists of a boolean expression followed by one or more statements.** |
| [**if...else statement**](http://www.tutorialspoint.com/cprogramming/if_else_statement_in_c.htm) | **An if statement can be followed by an optional else statement, which executes when the boolean expression is false.** |
| [**nested if statements**](http://www.tutorialspoint.com/cprogramming/nested_if_statements_in_c.htm) | **You can use one if or else if statement inside another if or else if statement(s).** |
| [**switch statement**](http://www.tutorialspoint.com/cprogramming/switch_statement_in_c.htm) | **A switch statement allows a variable to be tested for equality against a list of values.** |
| [**nested switch statements**](http://www.tutorialspoint.com/cprogramming/nested_switch_statements_in_c.htm) | **You can use one switch statement inside another switch statement(s).** |

**Program:**

#include<stdio.h>

#include<dos.h>

#include<conio.h>

#include<string.h>

void main()

{

static char temp[40];

char name[20];

char ch;

FILE \*fptr;

int i=0;

int k=0;

int count=0;

int count1=0;

clrscr();

printf("\n\tEnter file name: ");

gets(name);

fptr=fopen(name,"r");

if(fptr==NULL)

printf("fptr could not open");

ch=fgetc(fptr);

do

{

if(ch=='?')

{

count1++;

k=4;

}

else if((ch>=65&&ch<=90)||(ch>=97&&ch<=122))

k=3;

else

k=4;

switch(k)

{

case 3:

do

{

if((ch>=65&&ch<=90)||(ch>=97&&ch<=122))

{

temp[i]=ch;

i++;

}

else

{

temp[i]=0;

if(!strcmp(temp,"for")||!strcmp(temp,"while"))

{

count++;

}

else if(!strcmp(temp,"if")||!strcmp(temp,"else")||!strcmp(temp,"switch"))

{

count1++;

}

i=0;

break;

}

}while((ch=fgetc(fptr))!=EOF);

break;

default:

ch=fgetc(fptr);

}

}

while(ch!=EOF);

printf("\nNo. of loops: %d",count);

printf("\nNo. of decesion makings: %d",count1);

printf("\n\nEnd of file reached.");

getch();

}

**Output:**

Enter file name: sample\_program.c

Number of Loops in the program: 4

Number of Decision making statements in the program: 12

End of file reached.

**Experiment : 6**

**Title : Perform Black box testing on a C/C++ program.**

**Theory :**

**Black Box Testing:**

Black-box testing, also called behavioural testing, focuses on the functional requirements of the software. That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.

**Techniques:**

* Boundary value analysis
* Equivalence partitioning testing

**Program: (Binary Search)**

1. #include<stdio.h>

2. #include<conio.h>

3. void binarysearch(int[],int,int);

4. void main()

5. {

6. int d[10],n,i,b;

7. clrscr();

8. printf("\n\tEnter the value of n ");

9. scanf("%d",&n);

10. printf("\n\tEnter the elements in sorted order \n");

11. for(i=0;i<n;i++)

12. {

13. scanf("%d",&d[i]);

14. }

15. printf("\n\tPlease enter the number you want to search ");

16. scanf("%d",&b);

17. binarysearch(d,n,b);

18. getch();

19. }

20. void binarysearch(int d[], int n, int b)

21. {

22. int begin=0,end=n-1,mid;

23. while(end>=begin)

24. {

25. mid=(begin+end)/2;

26. if(d[mid]==b)

27. {

28. printf("\n\tNumber found at position %d",mid);

29. return;

30. }

31. else if(b<d[mid])

32. end=mid-1;

33. else

34. begin=mid+1;

35. }

36. printf("\n\tNumber not found");

37. }

**Test Cases: (Boundary Value Analysis Technique)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Test case No. | Type | For variable or construct  **(Optional)** | **n** | **D** | **b** | Desired O/P | Actual O/P |
| 1 | Boundary Value Analysis | Variable **n** | -32768 |  | 5 | Invalid array size | Number not found |
| 2 | Boundary Value Analysis | Variable: **n** | -32767 |  | 5 | Invalid array size | Number not found |
| 3 | Boundary Value Analysis | Variable: **n** | -1 |  | 5 | Invalid array size | Number not found |
| 4 | Boundary Value Analysis | Variable: **n** | 0 |  | 5 | Invalid array size | Number not found |
| 5 | Boundary Value Analysis | Variable: **n** | 1 | [3] | 3 | Number found at position 0 | Number found at position 0 |
| 6 | Boundary Value Analysis | Variable: **n** | 32766 | [1,2,3,….] |  | Invalid array size | Number found at position 4 |
| 7 | Boundary Value Analysis | Variable: **n** | 32767 | [1,2,3,…..] | 5 | Invalid array size | Number found at position 4 |
| 8 | Boundary Value Analysis | Variable: **n** | 11 | [1,2,3,5,9,25,27,89,90,92,95] | 5 | Invalid array size | Number found at position 3 |
| 9 | Boundary Value Analysis | Variable: **d** | 3 | [-32768, -32768 , -32768] | 5 | Out of bounds, Invalid array size | Number not found |
| 10 | Boundary Value Analysis | Variable: **d** | 3 | [-32767, -32767  , -32767] | 5 | Out of bounds, Invalid array size | Number not found |
| 11 | Boundary Value Analysis | Variable: **d** | 3 | [-1,-1,-1] | 5 | Out of bounds, Invalid array size | Number not found |
| Test case No. | Type | For variable or construct  **(Optional)** | **n** | **d** | **b** | Desired O/P | Actual O/P |
| 12 | Boundary Value Analysis | Variable: **d** | 3 | [0,0,0] | 5 | Number not found | Number not found |
| 13 | Boundary Value Analysis | Variable: **d** | 3 | [1,1,1] | 5 | Number not found | Number not found |
| 14 | Boundary Value Analysis | Variable: **d** | 3 | [32766,32766, 32766] | 5 | Number not found | Number not found |
| 15 | Boundary Value Analysis | Variable: **d** | 3 | [32767, 32767, 32767] | 5 | Out of bounds of integer | Number not found |
| 16 | Boundary Value Analysis | Variable: **d** | 3 | [2,3,5] | 5 | Number found at position 2 | Number found at position 2 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Test case No. | Type | For variable or construct  **(Optional)** | **n** | **d** | **b** | Desired O/P | Actual O/P |
| 17 | Boundary Value Analysis | Variable: **b** | 3 | [2,3,5] | -32768 | Out of bounds of integer | Number not found |
| 18 | Boundary Value Analysis | Variable: **b** | 3 | [2,3,5] | -32767 | Number not found | Number not found |
| 19 | Boundary Value Analysis | Variable: **b** | 3 | [2,3,5] | +32766 | Number not found | Number not found |
| 20 | Boundary Value Analysis | Variable: **b** | 3 | [2,3,5] | +32767 | Out of bounds of integer | Number not found |

**Experiment : 7**

**Title : Perform White box testing on a C/C++ program.**

**Theory :**

**White Box Testing:**

White-box testing, sometimes called glass-box testing is a test case design method that uses the control structure of the procedural design to derive test cases. Using white-box testing methods, the software engineer can derive test cases that:

(1) guarantee that all independent paths within a module have been exercised at least once,

(2) exercise all logical decisions on their true and false sides,

(3) execute all loops at their boundaries and within their operational bounds, and

(4) exercise internal data structures to ensure their validity.

**Techniques:**

* Basis path testing
* Condition testing
* Control loop testing
* Data flow testing

**Program: (Binary Search)**

1. #include<stdio.h>

2. #include<conio.h>

3. void binarysearch(int[],int,int);

4. void main()

5. {

6. int d[10],n,i,b;

7. clrscr();

8. printf("\n\tEnter the value of n ");

9. scanf("%d",&n);

10. printf("\n\tEnter the elements in sorted order \n");

11. for(i=0;i<n;i++)

12. {

13. scanf("%d",&d[i]);

14. }

15. printf("\n\tPlease enter the number you want to search ");

16. scanf("%d",&b);

17. binarysearch(d,n,b);

18. getch();

19. }

20. void binarysearch(int d[], int n, int b)

21. {

22. int begin=0,end=n-1,mid;

23. while(end>=begin)

24. {

25. mid=(begin+end)/2;

26. if(d[mid]==b)

27. {

28. printf("\n\tNumber found at position %d",mid);

29. return;

30. }

31. else if(b<d[mid])

32. end=mid-1;

33. else

34. begin=mid+1;

35. }

36. printf("\n\tNumber not found"); }

**Test Cases: (Loop Testing Technique)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Test case No. | Type | For variable or construct  **(Optional)** | **n** | **d** | **b** | Desired O/P | Actual O/P |
| 1 | Loop Testing | For **FOR** Loop | 0 |  | 5 | Invalid input | Number not found |
| 2 | Loop Testing | For **FOR** Loop | 1 | [3] | 5 | Number not found | Number not found |
| 3 | Loop Testing | For **FOR** Loop | 2 | [3,5] | 3 | Number found at position 0 | Number found at position 0 |
| 4 | Loop Testing | For **FOR** Loop | 5 | [3,4,12,101,199,607] | 5 | Number not found | Number not found |
| 5 | Loop Testing | For **FOR** Loop | 9 | [1,7,45,67,82,85,89,99,103] | 65 | Number not found | Number not found |
| 6 | Loop Testing | For **FOR** Loop | 10 | [2,9,11,65,87,99,105,134,189,200] | 200 | Number found at position 9 | Number found at position 9 |
| 7 | Loop Testing | For **FOR** Loop | 11 | [1,3,6,9,23,30,37,44,52,90,136] | 136 | Out of bound error | Number found at position 10 |
| 8 | Loop Testing | For **While** Loop | 0 |  | 5 | Invalid input | Number not found |
| 9 | Loop Testing | For **While** Loop | 1 | [3] | 3 | Number found at position 0 | Number found at position 0 |
| 10 | Loop Testing | For **While** Loop | 2 | [9,23] | 11 | Number not found | Number not found |
| 11 | Loop Testing | For **While** Loop | 2 | [5,7] | 3 | Number not found | Number not found |
| Test case No. | Type | For variable or construct  **(Optional)** | **n** | **d** | **b** | Desired O/P | Actual O/P |
| 12 | Loop Testing | For **While** Loop | 5 | [3,4,12,101,199] | 36 | Number not found | Number not found |
| 13 | Loop Testing | For **While** Loop | 9 | [1,7,45,67,82,85,89,99,103] | 65 | Number not found | Number not found |
| 14 | Loop Testing | For **While** Loop | 10 | [2,9,11,65,87,99,105,134,189,200] | 200 | Number found at position 9 | Number found at position 9 |
| 15 | Loop Testing | For **While** Loop | 11 | [1,3,6,9,23,30,37,44,52,90,136] | 136 | Out of bound error | Number found at position 10 |